Evaluation of Web Security Mechanisms Using Vulnerability & Attack Injection

José Carlos Coelho Martins da Fonseca, Marco Vieira, and Henrique Madeira

Abstract—In this paper we propose a methodology and a prototype tool to evaluate web application security mechanisms. The methodology is based on the idea that injecting realistic vulnerabilities in a web application and attacking them automatically can be used to support the assessment of existing security mechanisms and tools in custom setup scenarios. To provide true to life results, the proposed vulnerability and attack injection methodology relies on the study of a large number of vulnerabilities in real web applications. In addition to the generic methodology, the paper describes the implementation of the Vulnerability & Attack Injector Tool (VAIT) that allows the automation of the entire process. We used this tool to run a set of experiments that demonstrate the feasibility and the effectiveness of the proposed methodology. The experiments include the evaluation of coverage and false positives of an intrusion detection system for SQL Injection attacks and the assessment of the effectiveness of two top commercial web application vulnerability scanners. Results show that the injection of vulnerabilities and attacks is indeed an effective way to evaluate security mechanisms and to point out not only their weaknesses but also ways for their improvement.

Index Terms—Security, fault injection, internet applications, review and evaluation

1 Introduction

Nowadays there is an increasing dependency on web applications, ranging from individuals to large organizations. Almost everything is stored, available or traded on the web. Web applications can be personal websites, blogs, news, social networks, web mails, bank agencies, forums, e-commerce applications, etc. The omnipresence of web applications in our way of life and in our economy is so important that it makes them a natural target for malicious minds that want to exploit this new streak.

The security motivation of web application developers and administrators should reflect the magnitude and relevance of the assets they are supposed to protect. Although there is an increasing concern about security (often being subject to regulations from governments [1] and corporations [2]), there are significant factors that make securing web applications a difficult task to achieve:

1. The web application market is growing fast, resulting in a huge proliferation of web applications, based on different languages, frameworks, and protocols, largely fueled by the (apparent) simplicity one can develop and maintain such applications.
2. Web applications are highly exposed to attacks from anywhere in the world, which can be conducted by using widely available and simple tools like a web browser.
3. It is common to find web application developers, administrators and power users without the required knowledge or experience in the area of security.
4. Web applications provide the means to access valuable enterprise assets. Many times they are the main interface to the information stored in back-end databases, other times they are the path to the inside of the enterprise network and computers.

Not surprisingly, the overall situation of web application security is quite favorable to attacks [3], [4], [5]. In fact, estimations point to a very large number of web applications with security vulnerabilities [6], [7] and, consequently, there are numerous reports of successful security breaches and exploitations [8], [9]. Organized crime is naturally flourishing in this promising market, if we consider the millions of dollars earned by such organizations in the underground economy of the web [10], [11].

To fight this scenario we need means to evaluate the security of web applications and of attack counter measure tools. To handle web application security, new tools need to be developed, and procedures and regulations must be improved, redesigned or invented. Moreover, everyone involved in the development process should be trained properly. All web applications should be thoroughly evaluated, verified and validated before going into production.

However, these best practices are unfeasible to apply to the hundreds of millions of existing legacy web applications, so they should be constantly audited and protected by security tools during their lifetime. This is particularly relevant due to the extreme dynamicity of the security scenario, with new vulnerabilities and ways of exploitation being discovered every day. Clearly, security technology is not good enough to stop web application attacks and practitioners should be concerned with the evaluation and the assurance of their success [12]. In practice, there is a need for new ways to effectively test existing web application security mechanisms in order to evaluate and improve them.

- *J.C.C.M. da Fonseca is with the UDI of the Institute Polytechnic of Guarda and the CISUC. E-mail: josefonseca@ipg.pt.*
- *M. Vieira and H. Madeira are with the University of Coimbra and the Centre of Informatics and Systems of the University of Coimbra. Email: {mvieira, henrique}@dei.uc.pt.*

Manuscript received 17 June 2013; revised 29 Sept. 2013; accepted 3 Oct. 2013.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-2013-06-0164. Digital Object Identifier no. 10.1109/TDSC.2013.45

This paper proposes a methodology and a tool to inject vulnerabilities and attacks in web applications. The proposed methodology is based on the idea that we can assess different attributes of existing web application security mechanisms by injecting realistic vulnerabilities in a web application and attacking them automatically. This follows a procedure inspired on the fault injection technique that has been used for decades in the dependability area [13]. In our case, the set of "vulnerability" + "attack" represents the space of the "faults" injected in a web application, and the "intrusion" is the result of the successful "attack" of a "vulnerability" causing the application to enter in an "error" state [14]. In practice, a security "vulnerability" is a weakness (an internal "fault") that may be exploited to cause harm, but its presence does not cause harm by itself [15].

Conceptually, the attack injection consists of the introduction of realistic vulnerabilities that are afterwards automatically exploited (attacked). Vulnerabilities are considered realistic because they are derived from the extensive field study on real web application vulnerabilities presented in [16], and are injected according to a set of representative restrictions and rules defined in [17].

The attack injection methodology is based on the dynamic analysis of information obtained from the runtime monitoring of the web application behavior and of the interaction with external resources, such as the backend database. This information, complemented with the static analysis of the source code of the application, allows the effective injection of vulnerabilities that are similar to those found in the real world. In practice, the use of both static and dynamic analysis is a key feature of the methodology that allows increasing the overall performance and effectiveness, as it provides the means to inject more vulnerabilities that can be successfully attacked and discarded those that cannot.

Although this methodology can be applied to various types of vulnerabilities, we focus on two of the most widely exploited and serious web application vulnerabilities that are SQL Injection (SQLi) and Cross Site Scripting (XSS) [3], [6]. Attacks to these vulnerabilities basically take advantage of improper coded applications due to unchecked input fields at user interface. This allows the attacker to change the SQL commands that are sent to the database (SQLi) or through the input of HTML and scripting languages (XSS).

The proposed methodology provides a practical environment that can be used to test countermeasure mechanisms (such as intrusion detection systems (IDSs), web application vulnerability scanners, web application firewalls, static code analyzers, etc.), train and evaluate security teams, help estimate security measures (like the number of vulnerabilities present in the code), among others. This assessment of security tools can be done online by executing the attack injector while the security tool is also running; or offline by injecting a representative set of vulnerabilities that can be used as a testbed for evaluating a security tool.

The methodology proposed was implemented in a concrete *Vulnerability & Attack Injector Tool (VAIT)* for web applications. The tool was tested on top of widely used applications in two scenarios. The first to evaluate the effectiveness of the VAIT in generating a large number of realistic vulnerabilities for the offline assessment of security tools, in particular web application vulnerability scanners. The second to show how it can exploit injected vulnerabilities to launch attacks, allowing the online evaluation of the effectiveness of the counter measure mechanisms installed in the target system, in particular an intrusion detection system. These experiments illustrate how the proposed methodology can be used in practice, not only to uncover existing weaknesses of the tools analyzed, but also to help improve them.

The structure of the paper is as follows. The next section presents related research. Section 3 describes the proposed attack injection methodology, detailing its four stages. Section 4 presents the architecture of the VAIT prototype. Section 5 discusses several scenarios where the proposed methodology can be used and Section 6 describes the experiments and discusses the results. Finally, Section 7 concludes the paper.

2 Related Work

Fault injection techniques have been largely used to evaluate fault tolerant systems [18], [19]. The artificial injection of faults in a system (or in a component of the system) speeds up the occurrence of errors and failures, allowing researchers and engineers to evaluate the impact of faults on the system and/or the effects of potential error propagation to other systems. Fault injection also helps in estimating fault tolerant system measures, such as the fault coverage and error latency [18].

Fault injection techniques have traditionally been used to inject physical (i.e., hardware) faults [18], [19]. In fact, initial fault injection techniques used hardware-based approaches such as pin-level injection or heavy-ion radiation. The increasing complexity of systems has lead to the replacement of hardware-based techniques by software implemented fault injection (SWIFI), in which hardware faults are emulated by software. Xception [20] and NFTAPE [21] are examples of SWIFI tools.

The injection of realistic software faults (i.e., software bugs) has been absent from fault injection effort for a long time. First proposals were based on ad-hoc code mutations [22], [23], but more recent works focus on the injection of representative software faults based on comprehensive field studies on the most common types of software bugs [24].

The use of fault injection techniques to assess security is actually a particular case of software fault injection, focused on software faults that represent security vulnerabilities or may cause the system to fail in avoiding a security attack. Neves et al. proposed an Attack Injector Tool (AJECT) to support the discovery of vulnerabilities in network servers, specifically IMAP servers [25]. To attack the target system they used predefined test classes of attacks and some sort of fuzzing. Our approach automatically discovers places in the web application code that can be used to inject vulnerabilities using fault injection techniques and smart fuzzing to seamlessly attack them.

The industry uses fuzzing and mutation testing to automate penetration testing of web applications. They rely on web application vulnerability scanner tools that also generate reports compliant with security regulations (Sarbanes-Oxley, PCI-DSS, etc.). Some of the best known of such tools are HP WebInspect, IBM Watchfire AppScan, Acunetix web application security scanner and Web-Sphinx. In spite of their continuous development, these tools still have many problems related to the high number of undetected vulnerabilities and high percentage of false positives, as shown by several studies [26], [27]. To address these problems, it was proposed a method to benchmark these scanners [26]. The method starts by identifying all the points where each type of bug can be injected, then injecting the bug. Many of these bugs injected are vulnerabilities that can be used to test and compare the performance of the scanners.

The use of model checkers for security analysis was also proposed [28]. In this case, the vulnerability is injected by mutating the formal model of the web application. The model is also used to generate test cases that are used to attack the web application in a semi-automatic way.

The list of possible types of vulnerabilities affecting web applications is enormous, but XSS and SQLi are at the top of that list, accounting for 32 percent of the vulnerabilities observed [3], [6]. This is why we focus on those two important vulnerabilities, SQLi and XSS.

An SQLi attack consists of tweaking the input fields of the webpage (which can be visible or hidden) in order to alter the query sent to the back-end database. This allows the attacker to retrieve sensible data or even alter database records. An SQLi attack can be dormant for a while and only be triggered by a specific event, such as the periodic execution of some procedures in the database (e.g., the scheduled database record cleaning function).

A XSS attack consists of injecting HTML and/or other scripting code (usually Javascript) in a vulnerable webpage. It exploits the common utilization of the user input (without sanitizing it first) as a building part of a webpage. When this occurs, by tweaking the input, the attacker is able to change some of its functions, allowing him to take advantage of users visiting that webpage. This attack exploits the confidence a user (victim) has on the website, allowing the attacker to impersonate these users and even execute other types of attacks such as cross site request forgery (CSRF) [29]. The injection of XSS can also be persistent if the malicious string is stored in the back-end database of the web application, therefore potentiating its malicious effects in a much broader way.

A contribution to better understand the most common vulnerabilities in web applications was presented in a field study that classified 655 XSS and SQLi security patches of six widely used Linux, Apache, MySQL and PHP (LAMP) web applications [16]. LAMP is considered to be the most common stack of technologies used to build web applications and these types of applications are also prone to many vulnerabilities, namely XSS and SQLi. Both XSS and SQLi vulnerabilities result from poorly coded applications that do not properly check their inputs. One major conclusion of that study is that the most common type of vulnerabilities in web application code is by far, the "*Missing Function Call- extended*" (MFCE), with about 3/4 of all vulnerabilities found. Due to its relevance it was expanded into three sub-types, explained in Table 1 (see [16] for more details, other types and sub-types). This MFCE fault type represents vulnerabilities caused by an input variable that should have been properly sanitized by a specific function, which the programmer "forgot" to include in the code. Table 1 shows that sub-type A, originated by unchecked numeric fields, is the most relevant. This result is also corroborated by another work, this time referring only to SQLi vulnerabilities found in BugTraq SecurityFocus and presented by the open web application security project (OWASP) [30]. This study concludes that about half of SQLi vulnerabilities come from the exploitation of numeric fields.

TABLE 1
Missing Function Call—Extended (MFCS) Sub-Types

Sub-type	SQL (%)*	Description
A	64.25	Missing casting to numeric of one variable
B	4.15	Missing assignment of one variable to a custom made function
C	4.15	Missing assignment of one variable to a PHP predefined function

* *The values are refer to all the SQLi vulnerabilities analyzed in the field study detailed in [16].*

The methodology proposed in the present paper relies on the results of the field study presented in [16] to define the types of vulnerabilities to be injected (fault models), which match the most common types of vulnerabilities found in web applications in the field. These vulnerabilities are injected according to a set of representative restrictions and rules previously proposed in [17] and then attacked.

3 Vulnerability & Attack Injection Methodology

In this section we present the methodology for testing security mechanisms in the context of web applications. The methodology is based on the injection of realistic vulnerabilities and the subsequent controlled exploit of those vulnerabilities in order to attack the system. This provides a practical environment that can be used to test counter measure mechanisms (such as IDS, web application vulnerability scanners, firewalls, etc.), train and evaluate security teams, estimate security measures (like the number of vulnerabilities present in the code, in a similar way to defect seeding [31]), among others.

To provide a realistic environment we must consider true to life vulnerabilities. As mentioned before, we rely on the results from a field study presented in [16] that classified 655 XSS and SQLi security patches of six widely used LAMP web applications. This data allows us to define where a real vulnerability is usually located in the source code and what is the piece of code that is responsible for the presence of such vulnerability.

3.1 Overview of the Methodology

Our Vulnerability & Attack Injection methodology for SQLi and XSS can be applied to a variety of setups and technologies, but the following description uses as reference a typical web application, with a web front-end and

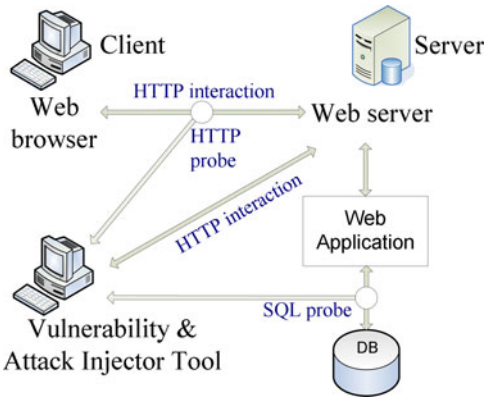

Fig. 1. VAIT in a typical setup.

an access to a back-end database to store the dynamic content and business data (Fig. 1).

The vulnerabilities are injected in the web application following a realistic pattern derived from [16]. The information about what was injected is fed to the injection mechanism in order to improve the attack success rate.

As shown in Fig. 1, the attack injection uses two external probes: one for the HTTP communication and other for the database communication. These probes monitor the HTTP and SQL data exchanged, and send a copy to be analyzed by the attack injection mechanism. This is a key aspect of the methodology to obtain the user interaction and the results produced by such interaction for analysis, so they can be used to prepare the attack. Therefore, the attack injection mechanism is aware of important inner workings of the application while it is running. For example, this provides insights on what piece of information supplied to a HTML FORM is really used to build the correlated SQL query and in which part of the query it is going to be inserted.

The entire process is performed automatically, without human intervention. For example, let's consider the evaluation of an IDS: during the attack stage, when the IDS inspects the SQL query sent to the database, the VAIT also monitors the output of the IDS to identify if the attack has been detected by the IDS or not. We just have to collect the final results of the attack injection, which also contains, in this case, the IDS detection output.

The automated attack of a web application is a multi-stage procedure that includes: *preparation stage, vulnerability injection stage, attackload generation stage, and attack stage*. These stages are described in the next sections.

3.2 Preparation Stage

In the preparation stage, the web application is interacted (crawled) executing all the functionalities that need to be tested (Fig. 2). Meanwhile, both HTTP and SQL communications are captured by the two probes and processed for later use. The interaction with the web application is always done from the client's point of view (the web browser).

The outcome of this stage is the correlation of the input values, the HTTP variables that carry them and their respective source code files, and its use in the structure of the database queries sent to the back-end database (for SQLi) or displayed back to the web browser (for XSS). Later on, in the attack stage, the malicious activity applied

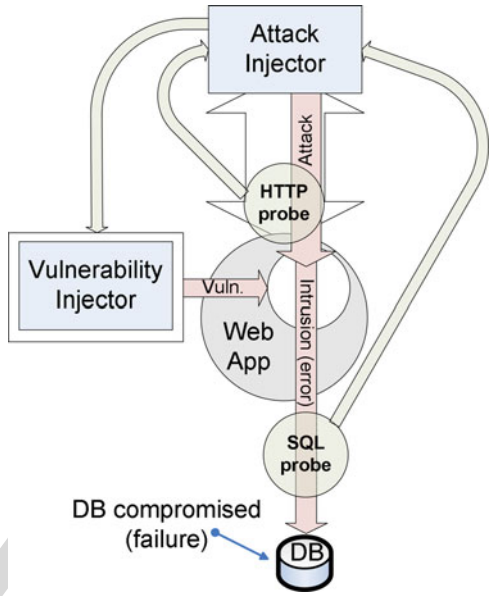

Fig. 2. Internal components of the VAIT.

is based on tweaking the values of the variables, which correspond to the text fields, combo boxes, etc., discovered in this preparation stage.

3.3 Vulnerability Injection Stage

It is in this vulnerability injection stage that vulnerabilities are injected into the web application. For this purpose, it needs information about which input variables carry relevant information that can be used to execute attacks to the web application. This stage starts by analyzing the source code of the web application files searching for locations where vulnerabilities can be injected (Fig. 2). The injection of vulnerabilities is done by removing the protection of the target variables, like the call to a sanitizing function. This process follows the realistic patterns resulting from the field study presented in [16]. Once it finds a possible location, it performs a specific code mutation in order to inject one vulnerability in that particular location. The change in the code follows the rules derived from [16], which are described and implemented as a set of Vulnerability Operators presented in [17].

The vulnerability operators are built upon a pair of attributes: the *Location Pattern* and the *Vulnerability Code Change*. The location pattern defines the conditions that a specific vulnerability type must comply with and the Vulnerability Code Change specifies the actions that must be performed to inject this vulnerability, depending on the environment where the vulnerability is going to be injected.

In order to clarify the concept of the vulnerability operators, let us analyze the following example. One of the location pattern restrictions for the missing function call extended subtype A (MFCE - A), is the search for the "`intval`"[1] PHP function when the argument is related to an input value (a value coming from the outside) and the result is going to be used in a SQL query string.

1. The "intval" PHP function returns the integer value of the argument. It returns -1 when the argument cannot be converted to an integer.

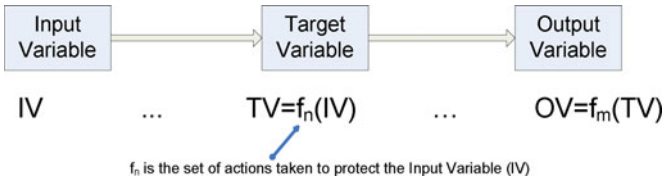

Fig. 3. Chain of variables from input to output of the web application.

Consider, for example, this sample piece of code: "`$id = intval($_GET['id']);`". If the variable "`$id`" is going to be used in a query, then the Vulnerability Code Change consists of removing the "`intval`" function from the source code in order to inject a vulnerability. As can be seen, by removing the function the resulting code becomes "`$id = $_GET['id'];`", which can be vulnerable to a SQLi attack. For example, by assigning the value "`15 or 1 = 1`" to the "`$id`" variable, the SQL query is executed without considering other constraints in the "`where`" condition. Recall that "`[anything] or 1 = 1`" is always true, therefore affecting every row of the query, which was not the intended behavior as coded by the developer of the application.

The vulnerability and attack injection uses both dynamic analysis and static analysis to gather the data needed to apply the vulnerability operators. This analysis obtains not only the input variables (IV) that will be part of an output variable (OV), but also the chain of variables in between. If the web application is secured, one of the variables in the chain is sanitized or filtered (Fig. 3). We call this variable our target variable (TV), because it is the one that the vulnerability injection stage will try to make vulnerable by removing or changing the protection scheme, according to the vulnerability operators. To inject a vulnerability using the Vulnerability Operators we need the information about the target variable and the code location (CL) where it is sanitized or filtered {TV, CL}.

In the preparation stage (based on the dynamic interaction executed by the crawler) we obtain the pairs $\{IV_{(dynamic\ analysis)}, OV_{(dynamic\ analysis)}\}$, which are the set of input variables ($IV_{(dynamic\ analysis)}$) whose values come from the HTTP interaction or the SQL communication and their mapping with output variables ($OV_{(dynamic\ analysis)}$). On the other side, the vulnerability injector tool performs a static analysis on the source code and finds the input variables ($IV_{(static\ analysis)}$) that are expected to be seen in the output ($OV_{(static\ analysis)}$) as part of the HTML response, SQL queries, etc. It also provides the target variable ($TV_{(static\ analysis)}$) and the code location ($CL_{(static\ analysis)}$) of the place in the file where the target variable is sanitized or filtered. Overall, the static analysis provides the following set of attributes: $\{IV_{(static\ analysis)}, OV_{(static\ analysis)}, TV_{(static\ analysis)}, CL_{(static\ analysis)}\}$.

This process of using dynamic and static results provides the best of both worlds to obtain the variables and the location where they are sanitized or filtered and the set of constraints given by the code location required by the vulnerability operators.

The correlation of variables resulting from both static and dynamic analysis originates a more precise set of locations where the vulnerability operators may be used. The outcome of this correlation is an improved collection of vulnerabilities that has a higher rate of exploitability by the attack injection mechanism. The data must be provided by the set of attributes that come from the static analysis $\{IV_{(static\ analysis)}, OV_{(static\ analysis)}, TV_{(static\ analysis)}, CL_{(static\ analysis)}\}$, but improved by the pair of attributes that come from the preparation stage $\{IV_{(dynamic\ analysis)}, OV_{(dynamic\ analysis)}\}$ (Fig. 4). It considers the data from the set of attributes $\{IV_{(static\ analysis)}, OV_{(static\ analysis)}, TV_{(static\ analysis)}, CL_{(static\ analysis)}\}$ but only whose pairs $\{IV_{(static\ analysis)}, OV_{(static\ analysis)}\}$ are equivalent to any of the $\{IV_{(dynamic\ analysis)}, OV_{(dynamic\ analysis)}\}$. The procedure to process the data from dynamic and static analysis to obtain the match outcome consisting of the pair of target variable and code location {TV, CL} needed to apply the vulnerability operators is exemplified in Fig. 5.

As a result of this vulnerability injection process, we obtain a copy of the original web application file with a

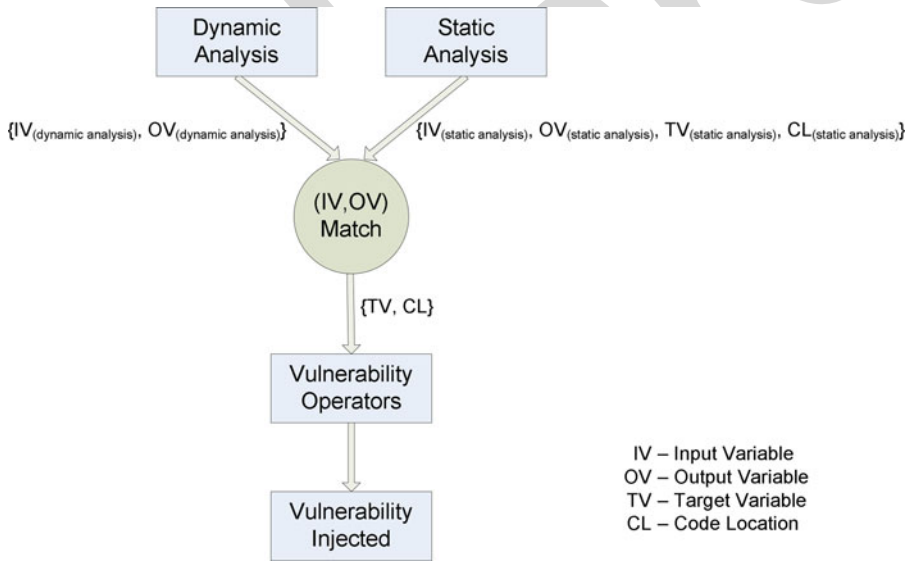

Fig. 4. Using data from dynamic and static analysis to apply the vulnerability operators and inject a vulnerability.

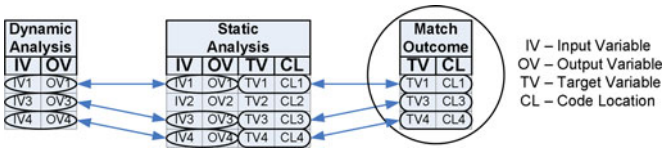

Fig. 5. Example of the use of data from dynamic and static analysis to obtain the match of target variable and code location for the vulnerability operators.

single vulnerability injected. This procedure can be automatically repeated until all the locations where realistic vulnerabilities can be injected are identified and all the corresponding vulnerabilities are injected, resulting in a set of files, each one with one possible vulnerability added (Fig. 6).

3.4 AttackLoad Generation Stage

After having the set of copies of the web application source code files with vulnerabilities injected we need to generate the collection of malicious interactions (attackloads) that will be used to attack each vulnerability. This is done in the attackload generation stage. The attackload is the malicious activity data needed to attack a given vulnerability. This data is built around the interaction patterns derived from the Preparation Stage, by tweaking the input values of the vulnerable variables.

The value that is assigned to the vulnerable variable, in order to attack it, results from a fuzzing process. In this process, the malicious value is obtained through the manipulation of the data provided by the good values of the vulnerable variable, the prefix (>,),',", ...) and the suffix (<,–,#,',", ...), the use of attackload strings and predefined functions (Fig. 7).

The fuzzing process consists of combining the available collection of prefixes, attackload strings and suffixes. For example, let us suppose that the variable may convey the value `John` and that its protection scheme has been removed by the Vulnerability Injection stage. In this case, one of the attackloads for SQLi assigns to the variable something like: "`John' +and+ 'A' = 'A`". In this attack string, the `John` is the known good value of the vulnerable variable, the ' is the prefix, the `+and+ 'A' = 'A` is the attackload string and there is no suffix (for this specific example). The + signs (they could as well be %20) are the URL encoded values of the space character, so the string can be used to build the malicious HTTP packet that will be sent to the web application by the attack injection mechanism.

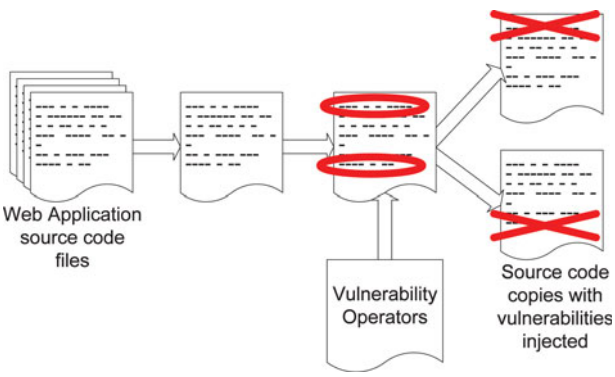

Fig. 6. The vulnerability injection stage.

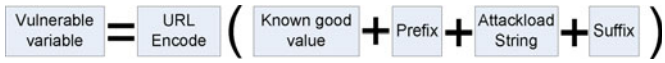

Fig. 7. Fuzzer generated malicious variable value.

This stage also generates the payload footprints that have a one to one relationship with the attack payloads. The payload footprints are the expected result of the attack. They can be the malicious SQL queries text sent to the database, for the case of an SQLi attack; or the HTML of the web application response, for the case of a XSS attack. These payload footprints are fundamental, since they are used to assess the success of the attack.

3.5 Attack Stage

In the attack stage, the web application is, once again, interacted. However, this time it is a "malicious" interaction since it consists of a collection of attack payloads in order to exploit the vulnerabilities injected. The attack intends to alter the SQL query sent to the database server of the web application (for the case of SQLi attacks) or the HTML data sent back to the user (for the case of XSS attacks).

The vulnerable source code files (from the vulnerability injection stage) are applied to the web application, one at a time. Once again the two probes for capturing the HTTP and SQL communications are deployed and the collection of attackloads is submitted to exploit the vulnerabilities injected (Fig. 2). The interaction with the web application is always done from the web client's point of view (the web browser) and the attackload is applied to the input variables (the text fields, combo boxes, etc., present in the webpage interface). At the end of the attack, we assess if the attack was successful. The detection of the success of the attack is done by searching for the presence of the payload footprint in the interaction data (HTTP or SQL communications) captured by the two probes. The process is repeated until all the injected vulnerabilities have been attacked.

4 Vulnerability & Attack Injector Tool

To demonstrate the feasibility of the proposed attack injection methodology we developed a prototype tool: the Vulnerability & Attack Injector Tool. For our research purposes the prototype currently focuses on SQLi, as it is one of the most important vulnerabilities of web applications today [3], [6]. Furthermore, SQLi is also responsible for some of the more severe attacks in web applications [8], [32], [33] as, nowadays, the most valuable asset of such applications is their back-end database. For this reason, we have chosen to implement first the SQLi type in our tool, although the XSS is quite similar in the key aspects.

The VAIT prototype targets Linux, Apache, MySQL and PHP web applications, which is currently one of the

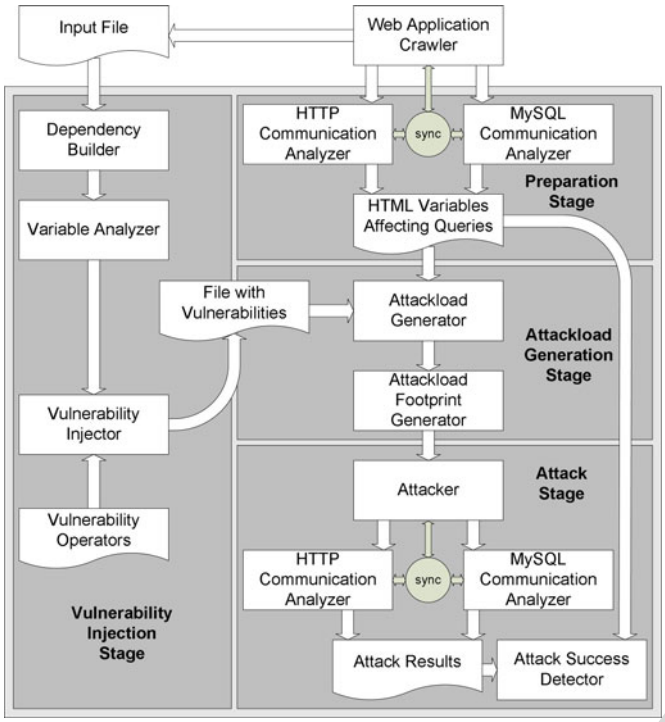

Fig. 8. Architecture of the VAIT.

most commonly used solution stack to develop web applications. Future improvements of the prototype may include other attacks types (e.g., XSS) and application technologies (e.g., Java).

The VAIT is an all-in-one application: it injects vulnerabilities into the web application code and attacks them in a seamlessly manner. As explained in the methodology description, the process of attacking the web application consists of (Fig. 8): the *preparation stage, the vulnerability injection stage, the attackload generation stage and the attack stage*. All this vulnerability and attack injection process is done with minimum human intervention. The VAIT is configured with the web application folder location. Then the preparation stage is executed while the web application is being interacted. At the end, the vulnerability injection stage automatically generates the vulnerabilities, followed by the attackload generation stage that generates the attack payloads. At this point, the attack stage can be executed to attack the vulnerabilities, collect the results and calculate the attack success.

During the Preparation Stage, the web application is executed. This interaction can be made either manually, by someone executing every web application procedure that should be tested, or automatically using an external tool, such as a *web application crawler*. During this interaction, the VAIT monitors the HTTP communication between the web browser and the web server and all the SQL communications going to and from the database server.

Monitoring is implemented using built-in proxies specifically developed for the HTTP and for the SQL communications. These proxies send a copy of the entire packet data traversing them through the configured socket ports to the *HTTP Communication Analyzer* and *MySQL Communication Analyzer* components. Proxies run as independent processes

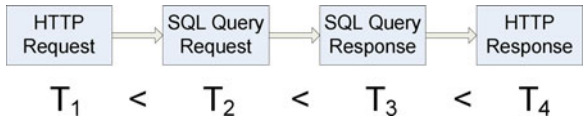

Fig. 9. Serialized sequence of actions processed by the Sync mechanism.

and threads, so they are relatively autonomous. To guarantee synchronization with other components of the VAIT, a *Sync mechanism* was also built-in (Fig. 8). The synchronism is obtained by executing each web application interaction in sequence without overlapping (i.e., without the common use of simultaneous threads to speedup the process) and gathering the precise time stamps of both the HTTP communication and respective SQL query. As shown in Fig. 9, the interaction starts with the client actor (the browser of the user of the web application) sending one HTTP request that may lead SQL query requests to be sent to the database server. Next, the database server responds to the SQL query requests and sends the response back to the web application server. Finally, the application server sends the HTTP response back to the client actor. When the HTTP and SQL proxies capture these serialized operations they also register their time stamps, which allows the Sync mechanism to group this distributed set of actions into meaningful cause-effect sequences (used to build the knowledge needed by the operation of the VAIT).

The information gathered by both proxies contains the structure of each webpage, the associated input variables, typical values and the associated SQL queries where these variables are used. During this interaction, the list of the web application files that are being run is also sent to the integrated *Vulnerability Injector* as input files. The vulnerability injector component is executed for each one, delivering the respective group of files with injected vulnerabilities.

Fig. 8 also shows the main components of the implementation of the Vulnerability Injection Stage. It comprises components to search for included files, analyze variables and finally inject vulnerabilities. The first component is the *Dependency Builder*. It searches recursively for the files that are included in the input file, which is the target PHP file where we want to inject the vulnerabilities. As in many other languages, in PHP programming, it is common to include a generic file inside another file, for reutilization purposes (this is done using one of the following statements: `include`, `include_once`, `require`, `require_once` [34]). During execution, the PHP interpreter processes the original file and its included files as a single block of code. When searching for locations where vulnerabilities may be injected, one should analyze the code in the same way the PHP interpreter does, thus including this dependency builder component.

The next component is the *Variable Analyzer*. Because SQLi vulnerabilities rely on vulnerable variables that can be exploited, we have to analyze all the variables that are used to build SQL queries. This component gathers all the PHP variables from the source code and builds a mesh of dependencies related to each other. Then, it searches for PHP variables present in SQL query strings. Using the

TABLE 2
Basic Attack Payload String Examples

Pre-defined attackload strings	Expected result of the attack
'	Change in the structure of the query. The query result is an error
or 1=1	Change in the structure of the query. The query result is the override of the query restrictions
' or 'a'='a	Change in the structure of the query. The query result is the override of the query restrictions
+connection_id()-connection_id()	Change in the query. The query result is 0
+1-1	Change in the query. The query result is 0
+67-ASCII('A')	Change in the query. The query result is 0
+51-ASCII(1)	Change in the query. The query result is 0
…	…

mesh created, the component is able to determine all the variables that are indirectly responsible for the SQL query. Both variables that are directly and indirectly responsible for SQLi are considered as a valid target to inject a vulnerability. This is important as one variable may be used only as input (POST or GET HTTP parameters) and the result is passed to another variable that is the one that is in the SQL query string. All the other variables are discarded.

The last component is the *Vulnerability Injector*. During execution, every location where the selected variables are found is tested with the conditions and restrictions of the vulnerability operators defined in [17], filtering those that are not applicable. The vulnerability operators, consisting of a set of location pattern and vulnerability code change attributes, as explained in Section 3.3, are derived from the detailed analysis of data presented in [16], which is partially summarized in Table 1.

The vulnerability injector component uses the vulnerability operator data and the result is the information about the mutation that has to be made in the source code in order to inject a particular vulnerability. Both the original source code and the mutated code (vulnerability injection code) are stored in the internal database of the VAIT for future consumption (e.g., during the execution of the Attack Stage).

Each of the vulnerable variables must be attacked and for that purpose, the *Attackload Generator* creates a collection of malicious interactions, according to the characteristics of the target variables. This attackload intends to inject unwanted features in the queries sent to the database, therefore performing SQLi. The collection of pre-defined attackload strings are based on the basic attacks presented in Table 2, but they can be extended covering other cases, like those presented by [35] or derived from field study data about real attacks [36]. Also, different database management systems have their own peculiarities on how they can be interacted and even different implementations of the SQL language have specific characteristics that can be exploited during a SQLi attack [37].

Every attack string is assigned to the vulnerable variable trying to create some sort of text that can penetrate the breach produced by the vulnerability injected (as shown previously in Fig. 7). Some tweaks are done to the attackload strings, such as encode some parts using the URL encoding function. The *Attackload Footprint Generator* component builds the collection of attackload footprints so that they have the data that is expected to be seen in the query, if the attack is successful.

The *Attack Stage* receives the files with vulnerabilities and the attackloads from the previous stage. All vulnerabilities are then executed from the web user perspective, one by one. To prevent bias from previous attacks, the web application files are copied from a safe location before injecting a vulnerability and the web application database is restored from a clean backup made before the start of the whole process. Using the generated attackload, the web application is automatically attacked. While the attack is being performed, the HTTP and SQL communications are monitored by the respective proxies and results are analyzed and stored in the attack injector tool internal database by the *HTTP Communication Analyzer and MySQL Communication Analyzer*, as explained before.

At the end it is necessary to verify if the attack was successful or not. This is done by the *Attack Success Detector* component. The attack is successful if, as a result of the execution of the attackload, the structure of the SQL query is altered [38]. This occurs when the attackload footprint is present in the query in specific conditions. Cases where the attackload footprint is placed inside a string variable of the SQL query are not considered, because usually a string can convey any combination of characters, numbers and signs. In the other cases, if it is possible to alter the structure of the query due to the attackload, then there is a successful SQLi attack.

One final remark about the VAIT is that it does not try to exploit the vulnerability in the sense of obtaining, altering, deleting, etc., sensible information from the web application database. It only tries to evaluate whether some particular instance of the web application (depending on the vulnerability injected) is vulnerable to such attacks or not. The VAIT also stores the SQL query string executed during the attack and the specific vulnerability exploited for later analysis. The output information given by the VAIT is the most important outcome and is a fundamental piece of data for enterprises and security practitioners. This data allows developers of the tool under assessment to correct the weaknesses discovered during the attack process. An example of an improvement of an IDS for databases that resulted from the output of the VAIT is presented in Section 6.2.

5 Attack Injection Utilization Scenarios

We envisage the following two scenarios as the most relevant utilizations of the proposed attack injection methodology and its VAIT tool:

1. *Inline*. The VAIT is executed while the security assurance mechanisms under evaluation are also being executed.
2. *Offline*. The VAIT is executed in advance to provide a set of realistic vulnerabilities for later use.

5.1 Inline Scenario

In the inline scenario, the VAIT can be used to evaluate tools and security assurance mechanisms, like IDS for databases, web Application IDS, web application firewalls and reverse proxies. For example, when assessing an IDS for databases (see Section 6.2 for a case study), the SQL probe should be placed before the IDS, so that the IDS is located between the SQL probe and the database (see Fig. 2 to locate the SQL probe and the database). During the attack stage, when the IDS inspects the SQL query sent to the database, the attack injector tool also monitors the output of the IDS to identify if the attack has been detected by the IDS or not. The entire process is performed automatically, without human intervention. The output of the VAIT also contains, in this case, the logs of the IDS detection. By analyzing the attacks that were not detected by the IDS, the security practitioner can gather some insights on the IDS weaknesses and, possibly, how the IDS could be improved. In addition to the case study presented in Section 6.2, this procedure has already been used to test five SQLi detection mechanisms [39].

5.2 Offline Scenario

In the offline scenario, the VAIT injects vulnerabilities into the web application and attacks them to check if they can be exploited or not. The outcome is the set of vulnerabilities that can, effectively, be attacked. They can then be used in a variety of situations, such as: to provide a testbed to train and evaluate security teams that are going to perform code review or penetration testing, to test static code analyzers, to estimate the number of vulnerabilities still present in the code, to evaluate web application vulnerability scanners, etc. It may also provide a ready to use testbed for web application security tools that can be integrated into assessment tools like the Moth [40] and projects like the Stanford Security Bench [41], or in web applications installed in honeypots prepared to collect data about how hackers execute their attacks. This gathers insights on how hackers operates, what assets they want to attack and how they are using the vulnerabilities to attack other parts of the system.

The offline scenario can also be applied to assess the quality of test cases developed for a given web application. For example, assuming that the test cases cover all the application functionalities in every situation, if the application code is changed (via vulnerability injection), the test cases should be able to discover that something is wrong. In situations where the test cases are not able to detect the modification, they should be improved and, maybe, the improvement can even uncover other unknown faulty situations.

As an example, let us consider the assessment of web application vulnerability scanners, used to test for security problems in deployed web applications (see Section 6.3 for a case study). These scanners perform black-box testing by interacting with the web application from the point of view of the attacker. In this scenario, the VAIT injects vulnerabilities and attacks them to see those that can be successfully attacked. These vulnerabilities are used, one by one, to assess the detection capability of the web application vulnerability scanner. This procedure can be used to obtain the percentage of vulnerabilities that the scanner cannot detect, and what are the most difficult types to detect. In this typical offline setup, the vulnerabilities can be injected one at a time (like in the case of vulnerability scanners) or multiple vulnerabilities at once (for the case of training security assurance teams, for example).

5.3 Attack Scenario Remarks

An attack can be considered successful if it leads to an "error" [14]. Obviously, the consequences of the attack (the "failure" and its severity) are dependent on the concrete situation, on what is compromised (credit card numbers, social security numbers, bank account information, passwords, emails, etc.), on how it is compromised (information disclosure, ability to alter the data or to insert new data, etc.) and on how valuable is the compromised asset (the value to the company, to the client from which the information belongs, to the companies operating in the same market, etc.) [10]. Although it is not a direct goal of the attack injection methodology presented here it can, however, provide important insights about security related issues allowing further analysis to obtain data about the consequences of the attack.

To avoid attacks, web application developers are currently reducing the number of error messages displayed to the user. This does not prevent SQLi attacks, but makes it harder to identify SQLi vulnerabilities using the black-box approach. However, after the vulnerability is found it is as easy to exploit as it was before. One consequence of this trend is an extraordinary increase in the false-positive and false-negative rates of black-box testing tools such as automatic web application vulnerability scanners [42], [27]. This also applies to other security mechanisms that use the same methodology, like the SQLmap sponsored by the OWASP project, for example [43]. The attack injection approach described in this chapter is quite immune to this countermeasure technique, because of the way the data used for the analysis is obtained: through the use of probes placed in different layers of the web application setup and correlating their data (e.g., HTTP and SQL interactions).

6 Experimental Evaluation and Results

To demonstrate the proposed VAIT we conducted three groups of experiments. In the first group, we injected vulnerabilities into three web applications to verify the quality of the vulnerabilities injected and the attack performance. In the second group, we tested an IDS for databases by using it inline with the VAIT. The goal was to evaluate the efficiency of the IDS in detecting the SQLi attacks performed by the VAIT. In the final group of experiments, we evaluated two top commercial web application vulnerability scanners regarding the detection of vulnerabilities that may be exploited by ad-hoc SQLi attacks.

For the evaluation experiments, we used Linux, Apache, Mysql and PHP web applications. The server runs Linux and the web server is Apache. This server hosts a PHP web application that uses a Mysql database. This software topology was chosen because it represents one of the most common technologies used to build custom web applications nowadays.

TABLE 3
Attack Injection Results of the Web Applications Analyzed

Web apps.	Files attacked	Code lines	Vuln. injected	Attacks	Attacks successful	Vulnerabilities attacked successfully
TikiWiki	tiki-editpage.php	904	3	84	34	3
	tiki-index.php	648	1	7	6	1
	tiki-login.php	305	3	21	0	0
	Total	1857	7	112	40 (36%)	4 (57%)
phpBB	search.php	1405	3	42	42	3
	login.php	224	1	21	21	1
	viewforum.php	694	1	7	7	1
	viewtopic.php	1210	5	84	84	5
	posting.php	1106	4	112	112	4
	Total	4639	14	266	266 (100%)	14 (100%)
MyRefs	edit_paper.php	310	27	525	61	20
	edit_authors.php	169	6	196	46	5
	Total	479	33	721	107 (15%)	25 (76%)
	Grand total	**6975**	**54**	**1099**	**413 (38%)**	**43 (80%)**

Three web applications were used in the experiments. The first is the groupware/content management system TikiWiki [44], which builds wikis (websites allowing users to contribute to them by adding and modifying their contents). It is widely used for building sites, such as the Official Firefox Support site and the KDE wiki. It was one of the finalists of the sourceforge.net 2007 for the most collaborative project award.

The second web application is the phpBB. It is a well-known LAMP web application and it has become the most widely used Open Source forum solution [45]. It is used by millions of users worldwide and won the sourceforge.net 2007 community choice award for best project for communications. It is also the forum module that is integrated into the phpNuke content management and portal web application. For these two applications (TikiWiki and phpBB) we bounded the attack surface only to the public sections, in order to limit the quantity of data that we had to analyze.

Lastly, there is a custom publication management web application called MyReferences. It was developed by a computer science PhD student for the management of PDF documents, and information about them such as the title, the conference, the year of publication, the document type, the relevance, and the authors. The information may be edited, queried and displayed.

Overall, the public section of TikiWiki has three files with 1,857 lines of code, phpBB has five files with 4,639 lines of code, whereas MyReferences has two files with 479 lines of code.

6.1 Vulnerabilities and Attacks Injected

The goal of this experiment was to validate the ability of the VAIT to inject vulnerabilities and also to exploit them to attack web applications, automatically. Towards this end, we wanted to know, on average, how many lines of code are necessary to be able to inject a vulnerability. Also, we wanted to know how many of those vulnerabilities can be successfully attacked. This gives a measure of the quality of the vulnerabilities injected, as it proves that they are indeed exploitable. Finally, we also wanted to know the effort needed to attack them and the success rate of these attacks. This gives a measure of the quality of the attacks. Besides being used as a sanity check of the VAIT, this data can also be used to help improve it in the future.

In the preparation stage, the gathering of the information about the web application pages and their links can be done manually or using a web crawler. In order to keep the same conditions for all the applications analyzed all the tests were done using the same web crawler, the one present in the Acunetix web vulnerability scanner. There are several web crawlers available nowadays [46], but only a few are able to insert values in the web application fields, such as the WebSphinx. For this purpose, the crawler presented in the WAVES framework can also be used [47] or the crawlers built in the commercial web application vulnerability scanners, which are usually very good in performing this task of website exploration.

The results of the attack injection in the target web applications are summarized in Table 3. The tool took approximately 11 minutes in the attack stage of the TikiWiki, 12 minutes in the phpBB and 4 minutes in the MyReferences. The vulnerabilities injected represent all the "Missing Function Call Extended" SQLi types that can realistically be injected into the files used in the experiments. As already stated, these vulnerabilities must comply with a restrictive set of rules in order to be considered realistic [17]. On average, the tool injected one vulnerability for every 129 lines of PHP code.

A collection of attackloads (see Table 2) was applied to each vulnerability injected and 38 percent of these attacks were successful. This measure of success comes from the presence of the attackload footprint in the SQL queries sent to the database.

We analyzed, one by one, each vulnerability injected that was not successfully attacked, in order to understand the reason why the attack was not successful. In five situations, belonging to the `edit_authors.php` file of the MyReferences web application the vulnerability was injected by removing an `intval` PHP function. By removing this function it is expected that the variable could be attacked injecting string values, such as "`or 1 = 1`". However, the affected variables are used inside strings formatted with the `%d` format, which also filters non-numeric variables. Therefore, this string formatting gives another level of protection preventing the attack to succeed through the supposedly vulnerable variable. In these situations, when the tool injects one vulnerability (by removing the code responsible for the sanitation of the variable) it leaves the other pieces of code still preventing the variable from being exploited. Recall that only a single vulnerability is injected at a time (even when multiple vulnerabilities can be injected in the same file). The reason is that we have no field study data supporting the realistic injection of more than one vulnerability at the same time.

All the other situations where it was not possible to attack the vulnerability, including the ones in `tiki-log-in.php` of the TikiWiki web application, are the result of a simplification in the prototype of the VAIT. This occurs when two variables with the same name are used in the same PHP file, although they are used in different blocks of code (they have a different scope). The VAIT can be tricked by this situation and, therefore, may try to inject a vulnerability in a place that has no relation with the right variable. In this case, the change in the code has no effect on the way the SQL query is built and, therefore, it is not an injection of a vulnerability. In the particular case tested, the problem was the use of a variable in a query and the use of an unrelated variable with the same name in a GET parameter of a HTML form. They are not related to each other as their scope of action is disjoint. This issue should be solved with the help of an improved PHP parser built into the VAIT.

The vulnerabilities that could not be attacked represent only 20 percent of all the vulnerabilities injected. Except for the particular cases explained before, the results show that the tool is effective in providing a sufficient number of realistic vulnerabilities in a web application and that these vulnerabilities can be successfully attacked. Furthermore, the output of some vulnerabilities that cannot be attacked is not a limitation of the methodology itself, but of simplifications of the variable analyzer component of the VAIT when evaluating the scope of PHP variables. However, most of these situations are going to be addressed by a new version of the PHP parser that is currently under development.

6.2 Case Study 1: IDS Evaluation

One possible use for the VAIT is the inline evaluation of security counter measures, such as an IDS for databases. An IDS is a very interesting tool, because it can defend the database from within, coping with new exploitation techniques that many times provide new means to overcome perimeter counter measures. In this case study, the IDS must be integrated with the VAIT, because the IDS output must be closely monitored during the attack stage.

From the previous experiment (Section 6.1) we know that the vulnerabilities injected can be successfully attacked. To evaluate the IDS we wanted to know its ability to detect the attacks to these vulnerabilities. This is done not only by obtaining the ratio of attacks detected (and not detected) by the IDS, but also by the false positives (false alarms). Both metrics are very important to characterize the IDS as they give a degree of assurance of what is expected to be detected (from the detection ratio) and the manual workload effort to do the screening process of all the alarms (from the false positive ratio). With the missing attacks and false alarms data we also wanted to know if the VAIT is able to provide enough information to help the developers to improve the IDS.

For this case study, we used an IDS for databases [48]. It can deal with Oracle and MySQL databases, but we only used the latter. This IDS implements an anomaly detection approach and includes a learning phase and a detection phase. Before initiating the attack injection, the IDS is trained with the target web application using the web crawler to execute the web application functions. After the training phase of the IDS, the VAIT is configured to operate together with the IDS and monitor its output.

The results of these experiments, for the three target web applications, are shown in Table 4. They show that the IDS was able to detect 99 percent of the attacks injected and missed only five of them (difference between the successful attacks and the attacks detected by the IDS). It also shows that, allied to the high detection rate of the IDS, there is also a high false positive rate.

The VAIT not only collects the results shown in Table 4, but it also gives all the details of the attacks, like the exact HTTP attack code, the target variable, the attackload used, the query sent to the database, etc. With all this information, developers and security practitioners can improve their security mechanisms and procedures. After this experiment, we analyzed why the IDS was unable to correctly detect all the attacks. Using the data collected by the VAIT we could replay these attacks while debugging the IDS. For example, this helped locate a defective function of the IDS, responsible for cleaning the SQL commands. There was one particular situation when processing the query structure that was not covered correctly, missing converting TAB characters to SPACE characters. Thanks to the VAIT, the bug is now fixed and this shows how the VAIT can also be used to improve security mechanisms. After fixing this bug, the IDS was able to detect all the attacks, although still providing some false positive values. These may be related to an insufficient learning period so, to be able to detect all good interactions as they are, the IDS must be trained for a longer period, until all the profiles are fully learned.

This experiment highlights the need to test security mechanisms considering realistic scenarios, which is one of the advantages of the VAIT. Further assessment of several SQL detection tools was already done using the proposed VAIT [39]. Some of the tools are widely used,

TABLE 4
Evaluation Results of the IDS

Web apps	Files attacked	Vuln. injected	Total attacks	Successful attacks	Attacks detected by the IDS	IDS false positives
TikiWiki	tiki-editpage.php	3	84	34	34	49
	tiki-index.php	1	7	6	6	1
	tiki-login.php	3	21	0	0	21
	Total	7	112	40	40 (100%)	71 (99%)
phpBB	search.php	3	42	42	42	0
	login.php	1	21	21	21	0
	viewforum.php	1	7	7	7	0
	viewtopic.php	5	84	84	84	0
	posting.php	4	112	112	112	0
	Total	14	266	266	266 (100%)	0 (0%)
MyRefs	edit_paper.php	27	525	61	61	294
	edit_authors.php	6	196	46	41	28
	Total	33	721	107	102 (95%)	322 (52%)
	Grand total	**54**	**1099**	**413**	**408 (99%)**	**393 (57%)**

like Apache Scalp, Snort or GreenSQL and others are from academia research, like the ACD Monitor and the IDS used in this case study. The results of the experiments highlighted the overall difficulty of these tools in detecting the attacks with a reasonable false positive rate (see [39] for details).

6.3 Case Study 2: Web Application Vulnerability Scanners Evaluation

In this case study, we evaluate another type of security tool: web application vulnerability scanners. These scanners are commercial tools used to audit the web application security from the point of view of the attacker as they try to penetrate the web application as a black-box (without accessing the source code). The scanners provide an easy and automatic way to search for vulnerabilities, avoiding the repetitive and tedious task of doing hundreds or even thousands of tests by hand for each vulnerability type. They can assess a myriad of security aspects such as XSS, SQLi, path traversal, file disclosure, web server vulnerabilities, etc. They use signatures of identified attacks of known web applications (and web application versions), but they can also test for ad-hoc XSS and SQLi vulnerabilities. In this study we used the HP WebInspect 7.7 (WebInspect) [49] and the IBM Watchfire AppScan 7.0 (AppScan) [50] commercial web scanners to test their ability to discover unreported SQLi vulnerabilities.

For the experiments with the scanners we wanted to know the percentage of vulnerabilities that they are able to detect. We also wanted to assess the relationship between the vulnerabilities detected by each scanner (to see if they are complementary to each other or if they are similar and detect the same set vulnerabilities). This data can be used not only to compare the scanners but also to help deciding if several scanners should be used, or if a manual analysis should also be performed, before deploying a web application.

The experiments are different from the ones conducted for the IDS. In this case, the VAIT is executed in advance (offline) for the three target web applications in order to identify the collection of vulnerabilities that could be attacked successfully. Then, for each vulnerability (one at a time), the web applications were tested with each scanner (also one at a time) and the results collected. Before running each scanner, the web application database was restored to prevent bias from previous experiments.

The complete results of the tests are detailed in Table 5. The number of SQLi vulnerabilities detected by the scanners is minimal. In fact, they were able to detect only 9 percent (WebInspect) and 7 percent (AppScan) of the vulnerabilities injected. The main reason for these poor results is that scanners heavily rely on the output of the web application (the HTML data the web browser receives from the web server) to detect vulnerabilities. However, the way web applications are built nowadays, hiding most of the error messages, make the task of identifying this type of vulnerabilities really difficult for black-box scanners. As a result, it is clear that the output of these scanners, when used to assess the security of an ad-hoc web application, cannot be the sole source used to assess the web application for vulnerabilities.

When collecting this data we also observed that there was only one vulnerability detected simultaneously by both scanners. All the others were only detected by a

TABLE 5
Overall Results of the Web Application Vulnerability Scanners

Web apps	Files attacked	Vuln. injected	Vulnerabilities attacked successfully	WebInspect	AppScan
TikiWiki	tiki-editpage.php	3	3	1	0
	tiki-index.php	1	1	0	0
	tiki-login.php	3	0	0	0
	Total	7	4	1 (25%)	0 (0%)
phpBB	search.php	3	3	0	1
	login.php	1	1	0	0
	viewforum.php	1	1	1	0
	viewtopic.php	5	5	1	1
	posting.php	4	4	0	0
	Total	14	14	2 (14%)	2 (14%)
MyRefs	edit_paper.php	27	20	1	0
	edit_authors.php	6	5	0	1
	Total	33	25	1 (4%)	1 (4%)
	Grand total	**54**	**43**	**4 (9%)**	**3 (7%)**

single scanner. The conclusion that different scanners find different vulnerabilities is confirmed by the results from other studies [27], so whenever possible several tools should be used simultaneously.

To improve the detection rate of SQLi, the scanners could use an approach similar to the one used by the VAIT: use a probe in the SQL communication path to gather data that can be sent back to the tool for analysis. In fact, an analogous scanning procedure that searches for an extensive collection of web application vulnerabilities is used by the AcuSensor technology from Acunetix [51].

7 Conclusion

This paper proposed a novel methodology to automatically inject realistic attacks in web applications. This methodology consists of analyzing the web application and generating a set of potential vulnerabilities. Each vulnerability is then injected and various attacks are mounted over each one. The success of each attack is automatically assessed and reported.

The realism of the vulnerabilities injected derives from the use of the results of a large field study on real security vulnerabilities in widely used web applications. This is, in fact, a key aspect of the methodology, because it intends to attack true to life vulnerabilities. To broaden the boundaries of the methodology, we can use up to date field data on a wider range of vulnerabilities and also on a wider range and variety of web applications.

To demonstrate the feasibility of the methodology, we developed a tool that automates the whole process: the VAIT. Although it is only a prototype, it highlights and overcomes implementation specific issues. It emphasized the need to match the results of the dynamic analysis and the static analysis of the web application and the need to synchronize the outputs of the HTTP and SQL probes, which can be executed as independent processes and in different computers. All these results must produce a single analysis log containing both the input and the output interaction results. The VAIT prototype focused on the most important fault type, the MFCE (vulnerabilities caused by a missing function protecting a variable), generating SQLi vulnerabilities. Although this fault type represents the large majority of all the faults classified in the field study and can be considered representative, other fault types can also be implemented, namely those that come next concerning their relevance.

The experiments have shown that the proposed methodology can effectively be used to evaluate security mechanisms like the IDS, providing at the same time indications of what could be improved. By injecting vulnerabilities and attacking them automatically the VAIT could find weaknesses in the IDS. These results were very important in developing bug fixes (that are already applied to the IDS software helping in delivering a better product). The VAIT was also used to evaluate two commercial and widely used web application vulnerability scanners, concerning their ability to detect SQLi vulnerabilities in web applications. These scanners were unable to detect most of the vulnerabilities injected, in spite of the fact that some of them seemed to easily be probed and confirmed by the scanners. The results clearly show that there is room for improvement in the SQLi detection capabilities of these scanners.

ACKNOWLEDGMENTS

This work was partially supported by the project "ICIS - Intelligent Computing in the Internet of Services" (CENTRO-07-ST24-FEDER-002003), co-financed by QREN, in the scope of the Mais Centro Program and European Union's FEDER, and by the project PEst-OE/EGE/UI4056/2011, financed by the Science and Technology Foundation.

REFERENCES

[1] USA, "Sarbanes-Oxley Act," 2002.
[2] *Payment Card Industry (PCI) Data Security Standard*, PCI Security Standards Council, 2010.
[3] S. Christey and R. Martin, "Vulnerability Type Distributions in CVE," Mitre Report, May 2007.
[4] S. Zanero, L. Carettoni, and M. Zanchetta, "Automatic Detection of Web Application Security Flaws," Black Hat Briefings, 2005.
[5] N. Jovanovic, C. Kruegel, and E. Kirda, "Precise Alias Analysis for Static Detection of Web Application Vulnerabilities," *Proc. IEEE Symp. Security Privacy*, 2006.
[6] J. Williams and D. Wichers, "OWASP Top 10," OWASP Foundation, Feb. 2013.
[7] IBM Global Technology Services "IBM Internet Security Systems X-Force 2012 Trend & Risk Report," IBM Corp., Mar. 2013.
[8] Verizon "2011 Data Breach Investigations Report," 2011.
[9] The Privacy Rights Clearinghousewww.privacyrights.org/databreach, Accessed 1 May 2013, Apr. 2012.
[10] M. Fossi, et al., "Symantec Internet Security Threat Report: Trends for 2010," Symantec Enterprise Security, 2011.
[11] M. Fossi, et al., "Symantec Report on the Underground Economy, Symantec Security Response," 2008.
[12] R. Richardson and S. Peters, "2010/2011 CSI Computer Crime & Security Survey," Computer Security Inst., 2011.
[13] D. Avresky, J. Arlat, J.C. Laprie, and Y. Crouzet, "Fault Injection for Formal Testing of Fault Tolerance," *IEEE Trans. Reliability*, vol. 45, no. 3, pp. 443-455, Sept. 1996.
[14] D. Powell and R. Stroud, "Conceptual Model and Architecture of MAFTIA," Project MAFTIA, Deliverable D21, 2003.
[15] V. Krsul, "Software Vulnerability Analysis," PhD thesis, Purdue Univ., West Lafayette, IN 1998.
[16] J. Fonseca and M. Vieira, "Mapping Software Faults with Web Security Vulnerabilities," *Proc. IEEE/IFIP Int'l. Conf. Dependable Systems and Networks*, June 2008.
[17] J. Fonseca, M. Vieira, and H. Madeira, "Training Security Assurance Teams using Vulnerability Injection," *Proc. IEEE Pacific Rim Dependable Computing Conf.*, Dec. 2008.
[18] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell, "Fault Injection and Dependability Evaluation of Fault-Tolerant Systems," *IEEE Trans. Computers*, vol. 42, no. 8, pp. 913-923, Aug. 1993.
[19] R. Iyer, "Experimental Evaluation," *Proc. IEEE Symp. Fault Tolerant Computing*, pp. 115-132, Special Issue FTCS-25 Silver Jubilee, 1995.
[20] J. Carreira, H. Madeira, and J.G. Silva, "Xception: Software Fault Injection and Monitoring in Processor Functional Units," *IEEE Trans. Software Eng.*, vol. 24, no. 2, Feb. 1998.
[21] D.T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R.K. Iyer, "NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors," *Proc. Computer Performance and Dependability Symp.*, 2000.
[22] J. Christmansson and R. Chillarege, "Generation of an Error Set that Emulates Software Faults," *Proc. IEEE Fault Tolerant Computing Symp.*, 1996.
[23] H Madeira, M. Vieira, and D. Costa, "On the Emulation of Software Faults by Software Fault Injection," *Proc. IEEE/IFIP Int'l Conf. Dependable System and Networks*, 2000.
[24] J. Durães and H. Madeira, "Emulation of Software Faults: A Field Data Study and a Practical Approach," *IEEE Trans. Software Eng.*, vol. 32, no. 11, pp. 849-867, Nov. 2006.
[25] N. Neves, J. Antunes, M. Correia, P. Veríssimo, and R. Neves, "Using Attack Injection to Discover New Vulnerabilities," *Proc. IEEE/IFIP Int'l Conf. Dependable Systems and Networks*, 2006.
[26] J. Fonseca, M. Vieira, and H. Madeira, "Testing and Comparing Web Vulnerability Scanning Tools for SQLi and XSS Attacks," *Proc. IEEE Pacific Rim Int'l Symp. Dependable Computing*, Dec. 2007.
[27] Ananta Security "Web Vulnerability Scanners Comparison," anantasec.blogspot.com/2009/01/web-vulnerability-scanners-comparison.html, accessed 1 May 2013, 2009.
[28] M. Buchler, J. Oudinet, and A. Pretschner, "Semi-Automatic Security Testing of Web Applications from a Secure Model," *Proc. Int'l Conf. Software Security and Reliability*, 2012.
[29] cgisecurity.net, www.cgisecurity.com/articles/csrf-faq.shtml#whatis, Dec. 2008.
[30] Sam NG. CISA, CISSP. SQLBlock.com, www.owasp.org/images/7/7 d/Advanced_Topics_on_SQL_Injection_Protection.ppt, 2006.
[31] S. McConnell, "Gauging Software Readiness with Defect Tracking," *IEEE Software*, vol. 14, no. 3, May/June 1997.
[32] SANS Institute isc.sans.org/diary.html?storyid=3823, accessed 1 May 2013, Jan. 2008.
[33] NTA, www.nta-monitor.com/posts/2011/03/01-tests_show_rise_in_number_of_vulnerabilities_affecting_web_applications_with_sql_injection_and_xss_most_common_flaws.html, Mar. 2011.
[34] The PHP Group pt.php.net, accessed 1 May 2013, Dec. 2007.
[35] W. Halfond, J. Viegas, and A. Orso, "A Classification of SQLi Attacks and Countermeasures," *Proc. Int'l Symp. Secure Software Eng.*, 2006.
[36] J. Fonseca, M. Vieira, and H. Madeira, "The Web Attacker Perspective-A Field Study," *Proc. IEEE Int'l. Symp. Software Reliability Eng.*, Nov. 2010.
[37] pentestmonkey.net/cheat-sheets, accessed 1 May 2013, pentestmonkey.net, 2009.
[38] G. Buehrer, B. Weide, and P. Sivilotti, "Using Parse Tree Validation to Prevent SQLi Attacks," *Proc. Int'l Workshop Software Eng. and Middleware*, 2005.
[39] I. Elia, J. Fonseca, and M. Vieira, "Comparing SQLi Detection Tools Using Attack Injection: An Experimental Study," *Proc. IEEE Int'l Symp. Software Reliability Eng.*, Nov. 2010.
[40] A. Riancho, "Moth, Bonsai-Information Security," www.bonsai-sec.com/en/research/moth.php, accessed 1 May 2013, 2009.
[41] B. Livshits, "Stanford SecuriBench," suif.stanford.edu/~livshits/securibench, Accessed 1 May 2013, 2005.
[42] J. Grossman, "SQLi, Eye of the Storm," *The Security J.*, vol. 26, pp. 7-10, 2009.
[43] B. Damele, "Sqlmap: Automatic SQLi Tool," sqlmap.sourceforge.net, Accessed 1 May 2013, 2009.
[44] TikiWiki, tikiwiki.org, Accessed 1 May 2013, Dec. 2008.
[45] phpBB, www.phpbb.com, accessed 1 May 2013, Dec. 2008.
[46] Java-source.net, 2008, java-source.net/open-source/crawlers, Accessed 1 May 2013.
[47] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai, "Web Application Security Assessment by Fault Injection and Behavior Monitoring," *Proc. Int'l Conf. World Wide Web*, pp. 148-159, 2003.
[48] J. Fonseca, M. Vieira, and H. Madeira, "Detecting Malicious SQL," *Proc. Conf. Trust, Privacy & Security in Digital Business*, Sept. 2007.
[49] HP, download.hpsmartupdate.com/webinspect, Accessed 1 May 2013, Sept. 2013.
[50] IBM, www-03.ibm.com/software/products/us/en/appscan, Accessed 1 May 2013, Sept. 2013.
[51] Acunetix "Finding The Right Web Application Scanner; Why Black Box Scanning Is Not Enough," www.acunetix.com/websitesecurity/rightwvs.htm, Accessed 1 May 2013, 2009.

José Carlos Coelho Martins da Fonseca received the PhD degree in informatics engineering from the University of Coimbra in 2011. Since 2005, he has been with the CISUC as a researcher. He teaches computer related courses in the Polytechnic Institute of Guarda since 1993. He is the author or coauthor of more than a dozen papers in refereed conferences. His research on vulnerability and attack injection was granted with the DSN's William Carter Award of 2009, sponsored by the IEEE Technical Committee on Fault-Tolerant Computing (TC-FTC) and IFIP Working Group on Dependable Computing and Fault Tolerance (WG 10.4).

Marco Vieira is an assistant professor at the University of Coimbra, Portugal, and an adjunct associate teaching professor at the Carnegie Mellon University. He is an expert on dependability benchmarking and his research interests also include experimental dependability evaluation, fault injection, security benchmarking, software development processes, and software quality assurance, subjects in which he has authored or coauthored more than 100 papers in refereed conferences and journals. He has participated in many research projects, both at the national and European level. Marco Vieira has served on program committees of the major conferences of the dependability area and acted as referee for many international conferences and journals in the dependability and databases areas.

Henrique Madeira is a full professor at the University of Coimbra, where he has been involved in the research on dependable computing since 1987. He has authored or coauthored more than 150 papers in refereed conferences and journals and has coordinated or participated in tens of projects funded by the Portuguese government and by the European Union. He was the Program co-chair of the International Performance and Dependability Symposium track of the IEEE/IFIP International Conference on Dependable Systems and Networks, DSN-PDS2004 and was appointed Conference Coordinator of IEEE/IFIP DSN 2008.

Query to the Author

Q1. Deleted Registration symbol in this article. Please check.